Precimonius: Tuning Assistant for Floating-Point Precision

Cindy Rubio-González[1], Cuong Nguyen[1], Hong Diep Nguyen[1], James Demmel[1], William Kahan[1], Koushik Sen[1], David H. Bailey[2], Costin Iancu[2], and David Hough[3]

[1]EECS Department, UC Berkeley, {rubio, nacuong, hdnguyen, demmel, wkahan, ksen}@cs.berkeley.edu
[2]Lawrence Berkeley National Laboratory, {dhbailey, cciancu}@lbl.gov
[3]Oracle Corporation, david.hough@oracle.com

ABSTRACT

Given the variety of numerical errors that can occur, floating-point programs are difficult to write, test and debug. One common practice employed by developers without an advanced background in numerical analysis is using the highest available precision. While more robust, this can affect program performance significantly. In this paper we describe a dynamic program analysis technique to find a lower floating-point precision that can be used in any part of a program. PRECIMONIOUS performs a search on the program variables trying to lower their precision subject to accuracy constraints and performance goals. The tool then recommends a type instantiation for these variables using less precision while producing an accurate enough answer without causing exceptions. We evaluate PRECIMONIOUS on a few widely used functions from the GNU Scientific Library. For most of the programs tested, PRECIMONIOUS is able to reduce precision, which results in performance improvements as high as 25%.

1. INTRODUCTION

Floating point arithmetic [12, 21] is used in applications from a wide variety of domains such as high-performance computing, graphics or finance. To minimize the chance of problems, developers without an extensive background in numerical analysis are likely to use the highest available precision throughout the whole program. Even experts may find it difficult to understand how sensitive the execution is to rounding error and consequently default to using the highest precision available. While more robust, this can increase the program execution time, memory traffic, and energy consumption. Ideally, a programmer would use no more precision than needed in any part of an application.

To illustrate the problem, consider the experiment performed by Bailey [2] with a program that computes the arc length of an irregular function, written in `Fortran`. An implementation using `double` precision computes a result whose error is about $2 \cdot 10^{-13}$, compared to a second more accurate implementation using `double double` precision, but is about 20x faster[1]. On the other hand, if `double double` precision is used only for *one* of the variables in the program, then the result computed is as accurate as if `double double` precision had been used throughout the whole computation while being almost as fast as the all-double implementation. Aside from the Bailey experiment [2], many other experiments [7, 8, 17] show that programs implemented in mixed precision compute a result of the same accuracy faster than when using solely the highest precision arithmetic.

Given the complexity of existing software frameworks, it might be prohibitively expensive or downright impossible for developers to manually tune their floating-point precision. In this paper, we present a tool called PRECIMONIOUS (short for "parsimonious with precision") to help automate this process. Our tool has been implemented using the LLVM [16] compiler infrastructure and it recommends a program variant that uses less precision and produces results that satisfy accuracy and performance constraints.

PRECIMONIOUS exposes to application developers an interface to specify the accuracy acceptance criteria. It takes as input a program annotated with the developer's accuracy expectation and it implements a dynamic program analysis to find a program variant that uses lower precision subject to performance constraints. Given the set of program variables and their types, the analysis performs a search trying to reduce their precision while satisfying the constraints. To guarantee the performance constraints, the implementation uses either static performance models or dynamic instrumentation with hardware performance counters. The latter allows for easy extensibility to performance metrics such as energy or memory traffic. The search is based on the *delta-debugging* algorithm [24], which exhibits an $n \log n$ average complexity and an n^2 worst-case complexity, where n is the number of variables to be tuned. PRECIMONIOUS also requires a representative set of inputs for any given error threshold.

We evaluated PRECIMONIOUS on 8 widely used functions from the GNU Scientific Library (GSL) [9] library and 2 programs described in [2]. For each function, we have generated a set of inputs intended to ensure good code coverage. We use running time as our performance metric. For most of the test programs, PRECIMONIOUS was able to detect lower precision type configurations when varying the degree of ac-

[1]The author runs the experiment on an Intel-based Macintosh system, uses `gfortran` as the compiler and the QD package [13] to implement `double double` type (approximately 31 digits).

curacy desired. In the cases in which our tool did not find a type configuration, it was because no configuration led to performance improvement. A *type configuration* is a mapping from each variable of interest to a type that is suggested by our tool. By using the precision suggested by our tool, which is found automatically, we observed speedup as high as 25% when running the programs transformed according to PRECIMONIOUS guidance. The main contributions of this paper are:

- We introduce a novel automated approach for recommending the optimal precision choice that leads to performance improvement in floating-point programs. The transformed program is guaranteed to use variables of lower precision and have better performance than the original program. To our knowledge, this is the first attempt to address this problem.
- We implement our algorithm in a tool called PRECIMONIOUS and demonstrate its effectiveness by evaluating our tool on 8 widely used functions from the GSL library and 2 programs described in [2]. We make PRECIMONIOUS, and all the data and results presented in this paper publicly available under BSD license at: `https://github.com/nacuong/precimonius`.

The rest of this paper is organized as follows. In Section 2, we present a motivating example and challenges. We then provide a detailed description of our solution in Section 3. We give implementation details and present our experimental evaluation in Section 4. The limitations of our approach are discussed in Section 5. Finally, we discuss related work in Section 6 before concluding in Section 7.

2. OVERVIEW

A motivating example to show the effect of floating-point precision on program performance has been introduced by Bailey in [2]. The task is to estimate the arc length of the following function over the interval $(0, \pi)$.

$$g(x) = x + \sum_{1 \le k \le 5} 2^{-k} \sin(2^k x)$$

The corresponding program sums $\sqrt{h^2 + (g(x_k + h) - g(x_k))^2}$ for $x_k \in [0, \pi)$ divided into n subintervals, where $n = 1000000$, so that $h = \frac{\pi}{n}$ and $x_k = kh$.

A straightforward implementation in `C` using `long double` precision is shown in Figure 1(a). When compiled with `gcc` and run on an Intel x86 system[2] this program produces the answer 5.795776322412856 (stored in variable `s1` on line 27).

If the program uses `double` precision instead, the resulting value would be 5.795776322413**11**, which is only correct up to 11 or 12 digits after the decimal point (compared to the result produced by the `long double` precision program).

Figure 1(b) shows an optimized implementation written in `C` by an expert numerical analyst. From the original eight floating-point variables appearing on the highlighted lines, the optimized program uses `long double` precision only for the variable `s1` on line 17. Six other variables have been lowered to `double` precision and one variable to `float`. The program produces the correct answer and runs 10% faster than the `long double` precision version. When replacing `long double` with even higher `double double` precision from the QD package [13], the same transformation remains valid [2] and the performance improvements can be as high as 20×. We note that `double double` arithmetic is considerably slower than `long double` arithmetic because it is done in software, and each operation requires a sequence of roughly 20 instructions.

For any program, manually finding such an optimized type configuration requires domain-specific knowledge combined with advanced numerical analysis expertise. While probably feasible for small programs, manual changes are prohibitive for large code bases.

Our proposed tool automates precision tuning of floating-point programs. We reduce the problem of precision tuning to determining which program variables, if any, can have their types changed to a lower precision while producing an answer within a given error threshold. We say such an answer is *accurate enough*. In addition to satisfying accuracy requirements, the changes also have to satisfy performance constraints, e.g. the transformed program should be at least as fast as the original.

The main challenge for any automated approach is devising an efficient search strategy through the problem space. In our case this is the set of program variables and their type assignments. For the code example, there are eight floating-point variables and we consider three different floating-point precisions. The search space contains 3^8 possible combinations.

2.1 Searching for Type Configurations

For a given program, there may exist one or more *valid* type configurations that (1) use less floating-point precision than the original program, and (2) produce an accurate enough answer.

Different search algorithms can be considered depending on what valid configuration we are interested in finding. A *global* minimum corresponds to the type configuration that uses the least precision which translates to the best performance improvement among all valid type configurations for a given program. A *local* minimum (called *1-minimal* below) is a valid type configuration for which lowering the precision of any one additional variable would cause the program to compute an insufficiently precise answer or violate the performance constraint (e.g. being slower than the original program). We define a *change set*, denoted by Δ, to be a set of variables that are mapped to the higher precision. A change set, Δ, applied to the program under consideration results in a program variant where only the variables in Δ are in higher precision.

Finding a Global Minimum: Finding the global minimum may require evaluation of an exponential number of type configurations. To be precise, we may be required to evaluate P^n configurations, where n is the number of floating-point variables in the change set and P is the number of levels of floating-point precision. The naïve approach which evaluates all possible type configurations by changing one variable at a time is infeasible for large codes.

Finding a Local Minimum: If we are interested in finding a 1-minimal valid configuration, a naïve approach would

[2]All floating-point types used in this paper conform to the IEEE 754-2008 standard as implemented on various x86 architecture. In particular, `long double` is implemented as the 80-bit extended precision type with 64 significant bits.

```c
long double fun( long double x ) {
  int k, n = 5;
  long double t1;
  long double d1 = 1.0L;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  long double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```

(a) Program in `long double` precision

```c
double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```

(b) Tuned program using mixed precision

Figure 1: Two implementations of the `arclength` program using different type configurations. The programs differ on the precision of all floating-point variables except for variable `s1`.

consist of removing from the change set, say Δ, one element at a time, where Δ initially consists of all floating-point variables in the program. This means the element removed can be in a lower precision, while all other elements are in the higher precision. If any of these change sets translates to a program that passes the accuracy and performance test, we recurs with this smaller set. Otherwise Δ is 1-minimal, and we can stop the search.

This algorithm is illustrated via Figure 2(a). Each rectangle box represents a change set under test. A gray band denotes the set of variables that are removed from the change set (thus can be allocated in lower precision). A cross means the change set results in an invalid configuration (the program either produces an insufficiently precise answer or violates a performance constraint), while a check means the change set results in a valid configuration. In this figure, the algorithm finds a 1-minimal valid configuration after 4 iterations (the one in the dotted circle). Notice that lowering the precision of any additional variable in this configuration would make it invalid. This is illustrated in the fourth iteration - all configurations created are invalid.

Delta-Debugging: The delta-debugging search algorithm [24] finds a 1-minimal test case with the average running time of $O(n \times log(n))$. The worst case running time is still $O(n^2)$ and it arises when each iteration results in the reduction of the change set by one element, which reduces to using a naïve algorithm.

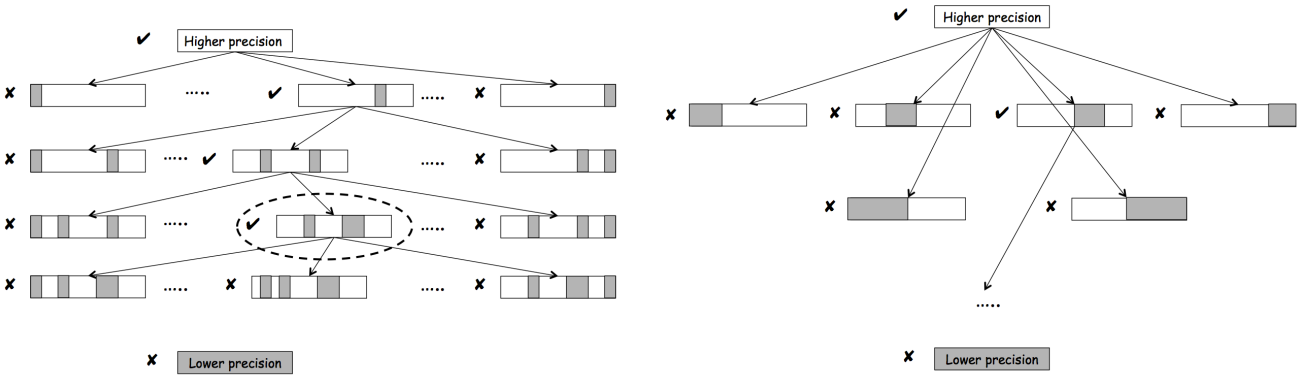
Informally, instead of making one type change at a time, the algorithm divides the change set in two and increases the number of subsets if progress is not possible. At a high level, the delta-debugging algorithm consists of partioning the change set Δ into almost equal size subsets $\triangle_1, \triangle_2, \cdots \triangle_n$, which are pairwise disjoint. The complement of a delta $\triangle_i$ is defined as $\bigtriangledown_i = \Delta - \triangle_i$. The algorithm starts by partitioning Δ into two sets ($n = 2$). After partitioning, the algorithm proceeds to examine each change set in the partition and their corresponding complements. The change set is reduced if a smaller failure inducing set is found, otherwise the partition is refined with $n = n * 2$.

This algorithm is illustrated via Figure 2(b). The original change set (higher precision rectangle box) is divided into two equal or almost equal size subsets. These two subsets result in two invalid configurations, so the partition granularity is increased to 4. The original change set is divided to 4 subsets accordingly, in which one of the subsets results in a valid configuration. The algorithm then recurses on this smaller subset.

3. PRECISION TUNING

We discuss the realization of our technique as a practical and effective tool for tuning the precision of floating-point programs, called PRECIMONIOUS. Figure 3 depicts its system architecture, which is built using the `LLVM` compiler infrastructure [16]. PRECIMONIOUS receives a C program, a test suite, and an accuracy requirement as input. It outputs a type configuration that, when applied to the original program, will result in a faster program that still produces an accurate enough answer without throwing exceptions.

PRECIMONIOUS consists of four main components: creating a search space for various program variants (Section 3.1), creating candidate program variants and eventually finding an optimal program variant using delta-debugging (Section 3.2), generating program variants for different type con-



(a) Quadratic number of variants

(b) Binary search on quadratic search space

Figure 2: Examples of search strategies

figurations (Section 3.3), and validating transformed programs (Section 3.4).

3.1 Creating the Search Space

The tool starts by creating a *search file* which consists of all variables whose precision needs to be tuned. The search file associates each floating-point variable with the set of floating-point types to be explored (e.g., `float, double`, and `long double`) for the variable. The input to this process is the program under analysis, in the format of `LLVM` bitcode. An excerpt of the search file for the `arclength` program in Section 2 is given below. The configuration states that the local variable `t1` in function `fun` can be either of type `float`, `double` or `long double` (for ease of presentation, we assume local variables in the same function have unique names).

```
1  ...
2  localVar: {
3    function: fun
4    name: t1
5    type: [float, double, long double]
6  }
7  ...
```

In the current implementation, the search file includes the local variables of all functions statically reachable from `main`. We only include global variables that are accessed in any of these functions. We include both scalars and arrays.

3.2 Finding an Optimal Configuration using Delta-Debugging

We employ a modified version of the delta-debugging algorithm [24] to find a type configuration which runs faster than the input program with accuracy requirement specified. To assess the profitability of a change, the implementation allows using both synthetic performance models as well as user supplied feedback such as hardware performance counters.

Synthetic Performance Models. To enable usage of performance models, the tool can generate instrumentation for basic operations. In our current prototype, we instrument all the floating-point instructions, including loads and stores. When each program variant is run, the dynamic count of these operations is gathered and it can be fed into performance models. We have experimented with several simple models none of which provided accurate predictor results in practice.

User Supplied Feedback. In order to accurately assess the desired performance metric during the program execution, the tool allows the user to provide her own instrumentation code. The search algorithm makes decisions based on these measurements. For all results presented in this paper, we used a performance metric obtained using hardware timers.

Search algorithm. Given a configuration search file, our goal is to find an optimal configuration which maps each variable to a single type and that exhibits lower cost. Initially, each variable in the search file is associated with a set of types. Our algorithm iteratively refines each of these sets of types until it consists of only one type. In each iteration, the algorithm considers a pair of types, the highest and second-highest precision available. It then determines the set of variables that need to be allocated in the highest precision. For these variables, the corresponding set of types are refined to contain only the highest precision type. These variables are then ignored in later iterations. For those variables that can be in the second-highest precision, the highest precision can be removed from their set of types and they are available as input to the next iteration.

Because our algorithm is based on delta-debugging search, with heuristic pruning, it is efficient in practice. To balance between the searching cost and the quality of result, we choose to search for a local minimum configuration. As shown in Section 4, we were able to find configurations with promising improvements on the workload considered.

Figure 4 shows our low cost configuration search algorithm, entitled LCCSearch. In this algorithm, a *change set* is a set of variables. The variables in the change set must have higher precision. The algorithm outputs a minimal change set, which consists of a set of variables that must be allocated in the higher precision (all other variables of interest can be in lower precision) so that the transformed program produces an accurate enough result and satisfies performance goal.

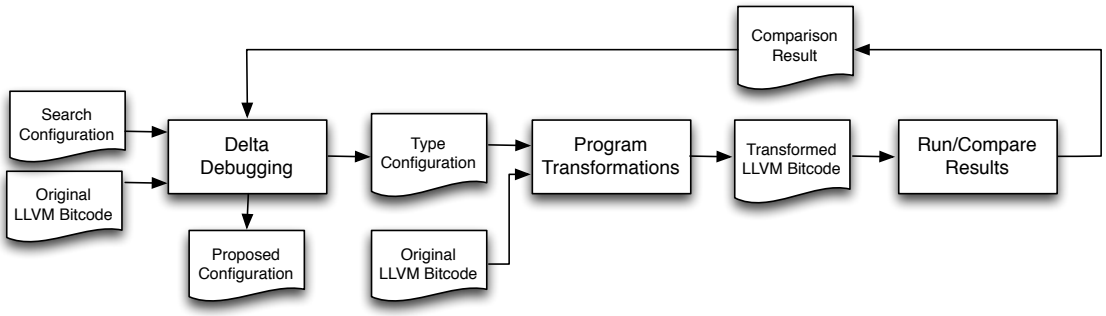


Figure 3: High-level framework components

Procedure LCCSearch
Inputs
P : target program
Δ : change set

Outputs
A minimal change set

Algorithm

```
1  div = 2
2  Δ_LC = Δ
3  for i in [1..div]:
4    Δ_i = Δ[((i−1)|Δ|)/div ... i|Δ|/div]
5    ∇_i = Δ \ Δ_i
6    if accurate(P, Δ_i) and cost(P, Δ_i) < cost(P, Δ_LC):
7      Δ_LC = Δ_i
8      div = 2
9    if accurate(P, ∇_i) and cost(P, ∇_i) < cost(P, Δ_LC):
10     Δ_LC = ∇_i
11     div = div−1
12 if Δ_LC ≠ Δ:
13   Δ = Δ_LC
14 else:
15   if div > |Δ|:
16     return Δ
17   else:
18     div = 2 ∗ div
19 goto 3
```

Figure 4: Lowest Cost Configuration Search Algorithm

The algorithm starts by dividing the change set Δ into two subsets of equal or almost equal size Δ_1 and Δ_2. It also creates the complement set of these subsets $\nabla_1 = \Delta \setminus \Delta_1$ and $\nabla_2 = \Delta \setminus \Delta_2$ (line 4-5). For each of these subsets, the algorithm creates the corresponding program variant (Section 3.3), checks whether the program variant produces an accurate enough result (Section 3.4) and records the one that has the lowest cost number (line 6-11). The function `accurate(P, Δ)` transforms the program `P` according to Δ and returns a boolean value indicating whether the transformed program is accurate enough. Similarly, the function `cost(P, Δ)` transforms the program `P` according to Δ and returns the cost of the transformed program. If a change set with the lowest cost exists, the algorithm recurses with that smaller change set (line 12-13 and 19); otherwise it restarts the algorithm with a finer-grained partition (line 17-19). In the special case where the granularity can no longer be increased, the algorithm returns the current Δ, which is a local minimum configuration (line 15-16).

3.3 Generating Program Variants

We automatically generate program variants that reflect the type configurations produced by our LCCSearch algorithm. We compile the original C program[3] to `LLVM` intermediate representation (`LLVM IR`) and apply a set of program transformations to reflect a given set of variable type assignments. We support scalar and array types. We currently do not support structures. The result of the transformation is a binary file (`LLVM` bitcode file) of the program whose variable types have been changed accordingly. The following subsections describe our program transformations given our intermediate representation of choice. Note that most `LLVM` instructions, including all arithmetic and logical operations, are in three-address form. An instruction in three-address form takes one or two operands, and produces a single result. Thus, our transformations are directly applicable to any other three-address form intermediate representation.

3.3.1 LLVM Intermediate Representation

`LLVM` (Low Level Virtual Machine) is a compiler infrastructure for program analysis and transformation. `LLVM` supports a language-independent instruction set and type system. `LLVM` has a static single assignment (SSA) based intermediate representation, meaning that each variable (called a typed register) is assigned only once. Instructions can only produce *first-class* types. `LLVM` defines eight such types: integer, floating point, pointer, vector, structure, array, label, and metadata.

Figure 5(a) shows excerpts of the bitcode file produced for the function `fun` from the `arclength` program of Figure 1(a). These excerpts correspond to allocating space for local variables `x`, `t1`, and `d1`, and the instructions corresponding to the statements on lines 6, 9 and 11 (all statements involving variable `t1`). The instruction `alloca` allocates memory in the stack. Other instructions include `load` to read from memory, `store` to store in memory, `fpext` and `fptrunc` to

[3]Because we are using the `LLVM IR`, we can also transform programs written in other languages for which an `LLVM` frontend is available (e.g., `Fortran`). However so far we have only transformed C programs.

cast floating-point numbers, and floating-point arithmetic operations such as `fadd`, `fsub`, `fmul` and `fdiv`. As mentioned earlier, each of these instructions is in three-address form, with no more than one operator on the right side of each assignment.

3.3.2 Program Transformations

We apply program transformations to change the type of variables according to the type configuration under consideration. We accomplish this by modifying the bitcode of the original program. For example, consider a type configuration that maps all floating-point variables to their original types except for variable `t1`, which is now suggested to be `double`. Figure 5(b) shows the transformed bitcode for the bitcode excerpts of Figure 5(a).

First, we replace `alloca` instructions to allocate the correct amount of memory for the variables whose type is to be changed. In this example, we allocate space for a `double` instead of a `long double` for variable `t1`. Second, we iterate through the uses of each of these instructions to identify other instructions that may need to be transformed. In this case, there are 5 instructions that use `t1` as an operand. These instructions are highlighted (in addition to the initial `alloca` instruction) in Figure 5(a). We define a set of program transformation rules to be applied depending on the instruction to be transformed. Table 1 shows a few sample transformation rules. For each transformed instruction, we iterate through its uses to continue to propagate the changes.

For example, consider the transformation of the second highlighted `store` instruction in Figure 5(a). This instruction is storing the result of an addition (register `%19`) into `t1`. The rule `Store-2` of Table 1 applies in this case because the target and the source of the `store` instruction have different types. The rule says that, in this case, the target should be downcast before it can be stored in the new variable `t1`. Thus, the transformed bitcode, shown in Figure 5(b), includes an additional `fptrunc` instruction right before the new transformed `store` instruction. Note that in the case of arithmetic operation instructions, we run a final pass to make sure each operation is performed in the precision of its highest-precision operand.

The resulting bitcode file verifies and can be run using `lli`, which directly executes programs in `LLVM` bitcode format. It is also possible to compile the bitcode to assembly, and use any back-end compiler such as `gcc`.

3.4 Logging and Checking Results

To determine whether a type configuration is valid, we run the transformed program and check for two criteria: correctness and performance.

First, we compare the result produced by the transformed program against the *expected result*. The expected result is the value (or values) obtained by running the original program on a given set of inputs. We take into account the error threshold provided by the programmer when comparing the results. Second, we measure running times for the original and transformed programs. Then, we determine whether the transformed program is at least as fast as the original program. This information is provided as feedback to our LCCSearch algorithm, which determines the new type configuration to be produced.

4. EXPERIMENTAL EVALUATION

As noted earlier, our implementation uses the `LLVM` compiler infrastructure [16]. We have written several `LLVM` passes in C++ to create search files, and to transform programs to have a given type configuration. We have implemented our LLCSearch algorithm in Python. Our result logger, result checker, and floating-point number generator have been implemented in C.

We present results for 8 programs that use the GNU Scientific Library (GSL) [9], the `arclength` program described throughout the paper, and a program that implements the Simpson's rule. We use Clang 3.0 to produce `LLVM` bitcode and a python-based wrapper [19] to build whole-program (or whole-library) `LLVM` bitcode files. We ran our experiments on an Intel Core i7-3770K 3.5Ghz Linux machine with 32GB RAM.

4.1 Experiment Setup

Precimonious can be used by developers with different levels of expertise in numerical analysis. In general, usage scenarios differ on the program inputs and threshold values selected for the analysis. A more experienced user might be more selective on the program inputs and threshold values to use. In the experiments presented in this section, we illustrate how a naïve user could employ our tool. We assume that the source code of the program is available, and the user knows which variable(s) will eventually store the result(s) produced by the program. We also assume that a test input set is not available.

For each program, we generate 1000 random floating-point inputs, and classify these inputs based on code coverage. We construct a representative test input set by selecting one input from each resulting group, thus attempting to maximize code coverage. Note that this test input set can be replaced or augmented by an existing input set. We annotate each program to log and check the results produced. This usually requires adding a few function calls to our logger and checker routines.

In our experiments, we use 4 different error threshold values, which indicate the number of accuracy digits required in each experiment. The non-expert user can compare the type configurations suggested by our tool for the various threshold values, and evaluate the trade-off between accuracy and speedup to make a decision. In general, the user can specify any threshold values of interest. Finally, we run each program thousands of times (or millions depending on their size) to ensure that the program runs long enough to obtain more reliable performance measurements.

4.2 Experiment Results

We ran our precision-tuning analysis on 8 programs that use the GSL library and 2 other programs (`arclength` and `simpsons`). Table 2 shows the number of variables in `float` (F), `double` (D) and `long double` (LD) precision in the original programs, and the type configurations suggested by our tool for each threshold value (10^{-4}, 10^{-6}, 10^{-8}, and 10^{-10}). Each error threshold indicates the number of accuracy digits. For example, 10^{-4} roughly means that the result is required to be correct up to 4 digits. The table also shows the number of configurations explored in each case by our LLCSearch algorithm, and its performance.

For example, our analysis took as little as 1 minute 3 seconds to find that we can lower the precision of all the variables in the program `roots` from `double` to `float` when the

```
define x86_fp80 @fun(x86_fp80) nounwind uwtable ssp {
  // allocating space for local variables
  %x = alloca x86_fp80, align 16
  %t1 = alloca x86_fp80, align 16
  %d1 = alloca x86_fp80, align 16
  // instructions to store initial values
  ...
  // t1 = x; (line 6)
  %2 = load x86_fp80* %x, align 16
  store x86_fp80 %2, x86_fp80* %t1, align 16
  ...
  // t1 = t1 + sin(d1 * x) / d1; (line 9)
  %10 = load x86_fp80* %t1, align 16
  %11 = load x86_fp80* %d1, align 16
  %12 = load x86_fp80* %x, align 16
  %13 = fmul x86_fp80 %11, %12
  %14 = fptrunc x86_fp80 %13 to double
  %15 = call double @sin(double %14) nounwind readnone
  %16 = fpext double %15 to x86_fp80
  %17 = load x86_fp80* %d1, align 16
  %18 = fdiv x86_fp80 %16, %17
  %19 = fadd x86_fp80 %10, %18
  store x86_fp80 %19, x86_fp80* %t1, align 16
  ...
  // return t1; (line 11)
  %24 = load x86_fp80* %t1, align 16
  ret x86_fp80 %24
}
```

(a) LLVM bitcode for original function

```
define x86_fp80 @fun(x86_fp80) nounwind uwtable ssp {
  // allocating space for local variables
  %x = alloca x86_fp80, align 16
  %t1 = alloca double, align 8
  %d1 = alloca x86_fp80, align 16
  // instructions to store initial values
  ...
  // t1 = x; (line 6)
  %2 = load x86_fp80* %x, align 16
  %3 = fptrunc x86_fp80 %2 to double
  store double %3, double* %t1, align 8
  ...
  // t1 = t1 + sin(d1 * x) / d1; (line 9)
  %11 = load double* %t1, align 8
  %12 = fpext double %11 to x86_fp80
  %13 = load x86_fp80* %d1, align 16
  %14 = load x86_fp80* %x, align 16
  %15 = fmul x86_fp80 %13, %14
  %16 = fptrunc x86_fp80 %15 to double
  %17 = call double @sin(double %16) nounwind readnone
  %18 = fpext double %17 to x86_fp80
  %19 = load x86_fp80* %d1, align 16
  %20 = fdiv x86_fp80 %18, %19
  %21 = fadd x86_fp80 %12, %20
  %22 = fptrunc x86_fp80 %21 to double
  store double %22, double* %t1, align 8
  ...
  // return t1; (line 11)
  %27 = load double* %t1, align 8
  %28 = fpext double %27 to x86_fp80
  ret x86_fp80 %28
}
```

(b) Transformed LLVM bitcode

Figure 5: LLVM bitcode excerpts of function `fun` from the `arclength` program of Figure 1

threshold value is 10^{-4}. The running time for the smaller program **`arclength`** was under a minute when using the same threshold value. Note that even the simple experiment of lowering the precision of all the variables in a program would be tedious and time consuming if done by hand. Our tool finds all these suggested type configurations without any manual intervention, and provides a listing of the new type assignment (if any) for each floating-point variable in the program. If desired, the user can run the modified `LLVM` bitcode of the program as well.

Our analysis runs in under 50 minutes for 38 out of the 40 experiments from Table 2 (10 programs and 4 different threshold values). The most expensive analysis was for the program **`fft`** with threshold 10^{-10}, which took 116 minutes 51 seconds. In this experiment, 663 type configurations were explored. Note that in this case there are 2^{22} or $4,194,304$ possible program variants.

Table 3 shows the speedup with respect to the original programs. PRECIMONIOUS finds at least one lower-precision type configuration that results in performance improvement for 7 out of 10 programs. The most significant improvement was observed for program **`blas`** with a speedup of 24.69% when the error threshold was set to 10^{-4} or 10^{-6}, followed by 15.07% for **`sum`**, and 13.05% for **`fft`**.

In general, we would expect that as the error threshold becomes larger, more program variables could be changed to a lower precision while improving performance. However, this is not necessarily true when the resulting configuration uses mixed precision. In some cases, the resulting type configuration might introduce many more casting operations, making the new program to actually run slower. Because our search constraints for valid configurations are both correctness and performance, there may be cases when a smaller threshold might prevent certain variables from being lowered, which might have caused the program to become more expensive due to castings, for example. Thus, a smaller threshold in some cases can lead to finding a type configuration that results in better program performance than a larger threshold.

For example, PRECIMONIOUS finds a type configuration that lowers 18 out of 19 floating-point variables for the program **`roots`** with threshold 10^{-6}. However, when we use a smaller error threshold of 10^{-8} or 10^{-10}, a different type configuration is found (also lowering 18 variables). In this case, the smaller threshold rules out invalid configurations that would have been valid for a larger threshold but would have had less performance improvement.

PRECIMONIOUS does not find a valid type configuration for any of the selected error thresholds for programs **`bessel`**, **`gaussian`**, and **`polyroots`**. In these cases, there are type configurations that use less precision while producing an accurate enough answer given those thresholds, however none of these type configurations lead to performance improvement. At least for **`bessel`**, it is well-known that there exist better and more efficient algorithms that could be used if we know beforehand that the computation can be performed in single precision. In the future, PRECIMONIOUS could be enhanced to incorporate this kind of domain knowledge.

Table 4 gives more details for the set of results corresponding to threshold value 10^{-6}. The table shows a comparison between the number of floating-point instructions executed by the original program and the suggested tuned program. In particular, we show the number of **`float`**, **`double`**, and **`long double`** loads, stores, arithmetic operations, comparison operations and casting instructions. We have omitted the programs for which a type configuration leading to better performance was not found. In general, we find that it

Table 1: Sample program transformation rules. Let $\Gamma = \{x : \tau \mid x \in \mathcal{V} \cup \mathcal{R}$ and $\tau \in \mathcal{T}\}$ be a finite set of type assignments for variables and registers. Γ is initialized with the type information given in the configuration file. Registers are initially untyped.

Name	Program Transformation		Type Rule	Description
`LOAD`	$r = v \Longrightarrow$	$r = v$	$\frac{\Gamma \vdash \mathtt{v} : \tau}{\Gamma \vdash \mathtt{r} = \mathtt{v} : \tau}$ `[Load]`	If $\Gamma \vdash v : \tau$ then $\Gamma \vdash r : \tau$
`STORE`	$v = r \Longrightarrow$	$v = r$	$\frac{\Gamma \vdash \mathtt{r} : \tau}{\Gamma \vdash \mathtt{v} = \mathtt{r} : \tau}$ `[Store-1]`	$\Gamma \vdash r : \tau$ and $\Gamma \vdash v : \tau$
	$v = r_1 \Longrightarrow$	$r_2 = (\tau)r_1$ $v = r_2$	$\frac{\frac{\Gamma \vdash \mathtt{r} : \tau'}{\Gamma \vdash \mathtt{r}_1 = (\tau)\mathtt{r} : \tau}}{\Gamma \vdash \mathtt{v} = \mathtt{r}_1 : \tau}$ `[Store-2]`	If $\Gamma \vdash v : \tau$ and $\Gamma \vdash r : \tau'$ then cast r to type τ before the store

is difficult to determine how different kinds of instructions contribute to improving or decreasing program performance.

Based on our results, we classify the programs under analysis into three categories: mixed precision, double precision and single precision. The following subsections explain each category and give some examples.

Mixed Precision: We find that for our test input setcertain inputs, the programs `simpsons`, `arclength`, `roots`, `rootnewt`, and `sum` indeed can use mixed precision to produce an accurate enough answer while improving program performance. For example, program `sum` takes the terms of a series and computes the extrapolated limit of the series using a Levin u-transform. There are 31 floating-point variables in this program. We find that we can lower the precision of 22 of these variables while improving program performance by 15.07%. Another example of mixed precision is illustrated by the `funarc` program described in Section 2, which led to a performance improvement of 9.84% to 11.16%.

Single Precision: It is possible for PRECIMONIOUS to determine that all variables in a program can be single precision. However, it is important to be aware that for certain mathematical functions, simply changing the precision of all variables might not be the best alternative to achieve better performance. Often, different algorithms are used for single precision, which can have a better performance. This is the case for programs such as `bessel` and `gaussian` (for which lowering the precision made the program slower, thus we did not suggest any type configuration). In this situation, the user might want to switch algorithms rather than lowering the precision of the double-precision algorithm. PRECIMONIOUS can still be helpful to indicate that all variables can be single precision, and hint the user to look for single-precision implementations of a given function. In the future, PRECIMONIOUS could be enhanced to switch implementations if sufficient domain-specific information is available.

Double Precision: PRECIMONIOUS can also help to determine whether a given program has already an optimal precision, preventing the user from making any program changes. The program `polyroots` illustrates this scenario. Our tool correctly finds that none of the variables involved in the computation can be changed to a lower precision.

5. LIMITATION

We note that our tool does not attempt to prove error bounds, or guarantee accurate answers for all possible inputs. This is a different and harder problem. Instead, we rely on the user to provide a set of representative inputs to make our decisions. If the user attempts to use our generated code on a much worse conditioned input, then we can make no guarantees; indeed even using the highest available precision everywhere may give the wrong answer in this case.

Furthermore, the transformation that PRECIMONIOUS currently performs is equivalent to the transformation of the program declaration at the source code level. It might be desirable to be able to change precision of intermediate variables introduced dynamically during execution, e.g. the same variable at different loop iterations can have different precision. This is doable at the LLVM bitcode level and subject to future work.

Finally, our evaluation involves two validity threats that we try to mitigate. Firstly, the sample of 10 programs in this study may be too small to yield statistically significant results. To reduce this threat, we select 8 programs from the widely-used and mature `GSL` library. The fact that we can still optimize these programs shows promise that developers can use our tool to optimize their early-developed and non-optimized programs. Secondly we may make some mistakes when mapping the type configuration found to the source code by hand, which might affect the performance speedup measurement. We performed a sanity check by comparing the numbers of load, store and arithmetic operation instructions in different level of precision between the program produced by PRECIMONIOUS and the program tuned by hand, and made sure that these numbers matched.

Table 3: Speedup observed after precision tuning

	Threshold			
Program	10^{-4}	10^{-6}	10^{-8}	10^{-10}
simpsons	9.99%	9.99%	9.99%	9.99%
arclength	9.84%	11.16%	10.35%	10.35%
bessel	0.00%	0.00%	0.00%	0.00%
gaussian	0.00%	0.00%	0.00%	0.00%
roots	7.09%	4.51%	6.79%	6.79%
polyroots	0.00%	0.00%	0.00%	0.00%
rootnewt	0.40%	4.45%	1.16%	0.47%
sum	15.07%	0.00%	0.00%	0.00%
fft	13.05%	13.05%	0.00%	0.00%
blas	24.69%	24.69%	0.00%	0.00%

Table 2: Performance analysis. The column *Original* gives the number of floating-point variables (`float` ***F*, `double` ***D***, and `long double` ***LD***) in the original program. For each selected threshold value, we give the type configuration (in number of variables per precision) found by our algorithm. *#Config* gives the total number of configurations explored by our LCCSearch algorithm. The running time of the analysis is given in (mm:ss).**

	Original			Threshold 10^{-4}					Threshold 10^{-6}				
Program	F	D	LD	F	D	LD	# Config	mm:ss	F	D	LD	# Config	mm:ss
simpsons	0	6	3	7	2	0	26	1:27	7	2	0	26	1:28
arclength	0	0	9	7	2	0	24	0:54	3	6	0	92	3:03
bessel	0	18	0	0	18	0	130	37:11	0	18	0	96	26:20
gaussian	0	52	0	0	52	0	201	16:12	0	52	0	161	12:29
roots	0	19	0	19	0	0	3	1:03	18	1	0	15	4:36
polyroots	0	28	0	0	28	0	336	43:17	0	28	0	161	21:42
rootnewt	0	12	0	8	4	0	61	16:56	5	7	0	25	6:08
sum	0	31	0	22	9	0	325	28:14	0	31	0	267	24:50
fft	0	22	0	22	0	0	3	1:16	22	0	0	3	1:17
blas	0	17	0	17	0	0	3	1:06	17	0	0	3	1:26

	Original			Threshold 10^{-8}					Threshold 10^{-10}				
Program	F	D	LD	F	D	LD	# Config	mm:ss	F	D	LD	# Config	mm:ss
simpsons	0	6	3	7	2	0	34	1:55	7	2	0	34	2:23
arclength	0	0	9	2	7	0	90	3:09	2	7	0	90	2:53
bessel	0	18	0	0	18	0	10	3:14	0	18	0	108	32:07
gaussian	0	52	0	0	52	0	1219	79:02	0	52	0	1435	86:32
roots	0	19	0	18	1	0	71	12:24	18	1	0	32	7:03
polyroots	0	28	0	0	28	0	244	28:50	0	28	0	302	31:31
rootnewt	0	12	0	7	5	0	57	9:48	7	5	0	23	5:32
sum	0	31	0	0	31	0	193	19:34	0	31	0	179	16:32
fft	0	22	0	0	22	0	423	78:10	0	22	0	663	116:51
blas	0	17	0	0	17	0	105	23:20	0	17	0	105	23:20

6. RELATED WORK

Our approach considers the problem of automatically finding the lowest precision that can be safely used in each part of a program. In recent work developed concurrently with our own, Lam et al. proposes a framework for automatically finding mixed-precision floating-point computation [15]. This work appears to be the most similar to ours. Their approach attempts to find double precision instructions that can be safely performed using single precision. They propose a brute-force based search using dynamic range analysis as heuristic. Their goal is to maximize the number of instructions that can be replaced to single precision.

PRECIMONIOUS uses a much more effective delta-debugging based search that can scale to a search space of at least 2^{50} configurations. Furthermore, it uses a more general performance model as the metric for searching. Our performance model can also be used to guide the search to maximize the number of instructions that can be replaced to single precision. However, this metric does not guarantee that the transformed program has better performance. Instead, we use running time as performance model and attempt to find a new program that runs faster than the original program. Finally, Lam et al. do not discuss the effectiveness of their tool in practice as we did in Section 4.

FloatWatch is a dynamic execution profiling tool for floating point programs which is designed to identify instructions that can be computed in a lower precision [6]. It works by first computing the overall range of values for each instruction of interest. Using this information, the tool recommends to use less precision if possible. Darulova and Kuncak also implemented a dynamic range analysis feature for the `Scala` language [10]. The approach uses interval and affine forms to represent the input, and examine how errors are magnified by each operation during execution. Their work might also be used for precision tuning purposes, by first computing a dynamic range for each instructions of interest and then tuning the precision based on the computed range, similar to *FloatWatch*. However, range analysis often incurs overestimates too large to be useful for precision tuning analysis.

Our work is also related to a large body of work on accuracy analysis. Benz et al. [4] presented a dynamic analysis approach for finding accuracy problems. Their approach computes every floating-point instructions side by side in higher precision. The higher precision computation is stored in a *shadow value*. If the differences between the original value and the shadow value become too large, their tool reports a potential accuracy problem. *FPInst* is another tool that computes floating point errors for the purpose for detecting accuracy problem [1]. It also computes a shadow value side by side, but it stores an absolute error in double precision instead. Lam et al. [14] propose a tool for detecting cancellation. Cancellation is detected by first computing the exponent of the result and the operands. If the exponent of the result is less than the maximum of those of the two operands, an cancellation has occurred. PRECIMONIOUS can complement accuracy analysis for debugging purposes in the following way. It can attempt to tune the set of potentially error-generating floating-point instructions to have higher precision until the accuracy problem goes away. The cost model could be changed to favor a configuration that requires the fewest changes.

Autotuning of codes to improve performance is a very large area of research, just a few citations being [5, 11, 18, 22, 23]. This previous work has however not tried to tune

Table 4: Number of instructions executed when the error threshold is 10^{-6}

Operations		roots original	roots tuned	rootnewt original	rootnewt tuned	fft original	fft tuned
Loads	F	0	1321	0	217	0	9106
	D	7786	6465	2506	2289	9106	0
	LD	0	0	0	0	0	0
Stores	F	0	641	0	148	0	4442
	D	3803	3162	1303	1155	4442	0
	LD	0	0	0	0	0	0
Arith Ops	F	0	217	0	0	0	4678
	D	2209	1992	898	898	4727	49
	LD	0	0	0	0	0	0
Comp Ops	F	0	78	0	0	0	0
	D	1593	1514	335	335	0	0
	LD	0	0	0	0	0	0
Castings	Trunc	0	207	0	69	0	28
	Ext	0	808	0	217	0	28

Operations		blas original	blas tuned	arclength original	arclength tuned	simpsons original	simpsons tuned
Loads	F	0	37805	0	15000001	0	9000015
	D	37805	0	0	21000001	7000006	7000008
	LD	0	0	36000002	0	9000001	0
Stores	F	13004	0	0	6000002	0	6000013
	D	13004	0	0	11000004	6000006	4000007
	LD	0	0	16000006	0	4000004	0
Arith Ops	F	0	24600	0	1	0	2000004
	D	24600	0	0	27000000	4000002	7000008
	LD	0	0	27000001	0	5000000	0
Comp Ops	F	0	0	0	0	0	0
	D	603	603	0	0	0	1000001
	LD	0	0	0	0	1000000	0
Castings	Trunc	0	0	6000001	5000000	2000001	6000010
	Ext	0	603	6000001	16000001	3000003	7000011

floating-point precision in the way this work does.

7. CONCLUSION

We have presented the first automated tuning algorithm for floating-point precision that identifies parts of the program that can be performed in lower precision. Our algorithm attempts to find a program that produces an accurate enough answer without exceptions and runs faster than the original program. We implemented our algorithm in an efficient and publicly available tool called PRECIMONIOUS. Initial evaluation on 8 programs using the `GSL` library shows encouraging results: we are able to discover precision configurations that result in performance improvements as high as 25%.

In the future we would like to apply our technique to a wider range of programs to gain better statistical results. We would also like to combine our tool with test generation techniques such as concolic testing [3, 20], so that, in the absence of a test suite, we can generate a more representative input set, making our tuning recommendations more stable across different inputs. Finally, we can generalize our technique to support not only variable precision but also operation, language construct and algorithm choices, which is also an interesting future research direction.

The PRECIMONIOUS source code, and all the data and results presented in this paper are available under BSD license at `https://github.com/nacuong/precimonius`.

8. REFERENCES

[1] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst, 2008. URL `http://www.freearrow.com/downloads/files/fpinst.pdf`.

[2] D. H. Bailey. Resolving numerical anomalies in scientific computation, 2008. URL `http://www.davidhbailey.com/dhbpapers/numerical-bugs.pdf`.

[3] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 549–560. ACM, 2013.

[4] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 453–462. ACM, 2012.

[5] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[6] A. W. Brown, P. H. J. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, pages 6–16, 2007.

[7] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Exploiting mixed precision floating point hardware in scientific computations, 2007.

[8] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):17:1–17:22, July 2008. doi: 10.1145/1377596.1377597. URL `http://doi.acm.org/10.1145/1377596.1377597`.

[9] G. P. Contributors. GSL - GNU scientific library - GNU project - free software foundation (FSF). http://www.gnu.org/software/gsl/, 2010. URL `http://www.gnu.org/software/gsl/`.

[10] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 325–344. ACM, 2011.

[11] M. Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, May 1999.

[12] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[13] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, pages 155–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=872021.872445`.

[14] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. In *1st International Workshop on High-performance Infrastructure for Scalable Tools*, 2011.

[15] M. O. Lam, B. R. de Supinski, M. P. LeGendre, and J. K. Hollingsworth. Automatically adapting programs for mixed-precision floating-point computation. Posters and Electronic Posters, International Conference for High Performance Computing, Networking, Storage and Analysis, 2012.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002. doi: 10.1145/567806.567808. URL `http://doi.acm.org/10.1145/567806.567808`.

[18] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[19] T. Ravitch. LLVM Whole-Program Wrapper @ONLINE, Mar. 2011. URL `https://github.com/travitch/whole-program-llvm`.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[21] I. C. Society. IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, Aug. 2008. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935`.

[22] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.

[23] C. Whaley. Automatically Tuned Linear Algebra Software (ATLAS). math-atlas.sourceforge.net, 2012.

[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.